Error Locating: Degree Constraints

by

Christopher Dennis

Bachelor of Science, University of Guelph, 2010

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master's of Science

Toronto, Ontario, Canada, 2016

©Christopher Dennis 2016

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

Abstract

Error Locating: Degree Constraints

Master's of Science 2016

Christopher Dennis

Applied Mathematics

Ryerson University

Error graphs are a useful mathematical tool for representing failing interactions in a system. This representation is used as the basis for constructing an error locating array (ELA). However, if too many errors are present in a given error graph, it may not be possible to locate all interactions. We say that a graph is *locatable* if an ELA can be built. Bounds on the total size of an error graph are known, bounds on the degree an error graph can have have not been considered. In this thesis we explore the maximum degree an error graph may have while still guaranteeing its locatability. We consider special cases for 3 and 4 partite error graphs as well as developing bounds on the degree of a general error graph. We describe a linear time algorithm which can be used to generate tests which have at most one failing interaction.

Acknowledgments

Firstly, I would like to thank my supervisor Dr. Peter Danziger, for providing me with the opportunity to study this interesting subject. Although I had not been in an academic environment for many years and did not have an undergraduate degree in math, he inspired me to work hard and fulfill my potential. Thanks to his patience, guidance, insights and careful attention to detail, my thesis has become a work I can truly be proud of. I have been very fortunate to have a supervisor who always gave his student his full attention no matter how busy he got.

I would like to thank my examiners Dr. Pawel Pralat and Dr. Eric Mendelsohn, for their careful review of my work as well as their thoughtful comments and contributions to the content of this thesis. As well I would like to thank Dr. Jean-Paul Pascal for agreeing to chair my defense committee.

For their financial support I would like to thank Ryerson university for providing me with many opportunities to be a graduate assistant. As well as the Department of Mathematics for admitting me to the Master's program in the first place.

I would also like to thank my fellow students who supported me during the completion of this degree. My unique circumstances during this time placed

demands on them that went far beyond what one could normally expect. However, fortunately for me there are still some who will show virtue beyond what could reasonably be asked of them. For that I will always be grateful.

My friends Palmer Jarvis and Steven Tomuak have always provided me with enough stimulating conversation and helpful distraction to keep me relatively sane throughout this whole process. It is wonderful to have such good friends who can keep me mentally sharp no matter the circumstances.

I can't possibly thank my family Mom, Dad, Spike and Ben enough for all their support during the many trials this degree presented me with. They have always been there for me when I needed them the most. My Uncle John Stechly and his wife Christine for their invaluable assistance during some difficult times. My cousin Jason White and his wife Maribel for making sure I ate a properly cooked meal at least occasionally, no matter how busy I became.

Dedication

I would like to dedicate this thesis to my Grandfather Leslie Dennis. His strength, wisdom and perseverance has always been something I have aspired to. Unfortunately he did not live to see me complete this work, but I know he would have been proud of what I have accomplished.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Combinatorial design theory is a study which concerns itself with taking a finite set of elements and arranging them according to a given set of rules. Typically we look to create subsets which satisfy certain properties. Different designs will select for different properties, so we can model many real world problems as design problems. Sometimes a design problem can be constructed in terms of graph theory, this can be important for making new insights into the original design problem. In this thesis we consider just such a case, where a covering problem is reformulated as a graph design problem.

In Section 1.1 we will cover the basics of the testing problem, in addition we will provide an example and discuss some practical considerations that will give insight into the direction our research should take. In Section 1.2 we give

some background terminology, particularly some specific graph theory terms which will be useful for our discussion. In Section 1.3 we will discuss some terms particular to covering type problems and introduce covering arrays. In Section 1.4 we introduce the concept of locating interactions within a given TP, particularly from a graph theory viewpoint as well as the fundamental concepts or error locating array and locatability. Much of this is due to the work of Martínez, Moura, Panario and Stevens [15]. In Section 1.5 we discuss the work of Colbourn and McClary [3], specifically the nature of *locating* and *detecting* arrays. Finally a brief summary of the thesis is provided in Section 1.6.

1.1 The Testing problem

Modern technological systems are typically a complex set of interacting components. It may be possible to determine that these components are not defective in and of themselves but it is possible that when paired with another component they will cause a fault. If a single defective component can be thought of as being "one dimensionally" defective, one might think of these problems as "defectiveness in higher dimensions". Locating these failures then becomes an important task in order to ensure that the final product will be functional. Most devices are a combination of sub-components which must all work in tandem for the piece of technology to function. Although

defects in any single component may be detected, modern devices may have a dizzying array of possible configurations. We assume that individual components have been tested for defects and consider the more difficult problem of detecting detrimental interactions between two or more otherwise functional components. This thesis will consider this possibility by examining pairwise testing through the lens of a combinatorial object. Before we get into the actual problems we must first establish a vocabulary for the testing problem, starting with the testing problem itself.

Definition 1.1.1. A **testing problem** is defined as a system with k components, hereafter called **factors**. Each factor $i \in \{1, \ldots, k\}$ has size g_i. A **test** is defined as a k-tuple, S, with one element from each factor. We designate a given testing problem as $\boldsymbol{TP(k; g_1, g_2, \ldots, g_k)}$, if all $g_1 = g_2 = \ldots = g_k = g$ we write $\boldsymbol{TP(k; g)}$.

For simplicity we typically refer to the general testing problem as simply TP, without the part number or size of the individual parts.

We must also introduce some terminology surrounding Testing Problems. An **alphabet** is defined as a set of g elements. In our case, each element from a factor will represent a different **level** of a component that we are testing. By convention this set is numerically represented as $\{0, \ldots, g-1\}$. In most cases that we consider, the size of the alphabet g is presumed to be constant across all k factors.

The testing problem reaches its apotheosis when the actual tests are conducted. So it is helpful to define what a test is in the context of the testing problem. A test is run with one element from each factor from a given $\boldsymbol{TP(k; g_1, g_2, \ldots, g_k)}$, creating a k-tuple. It will return either a **pass** in which case the test contained no failing interactions or a **fail** in which case at least one failing interaction is present. If a test passes it may be referred to as a **passing row**.

This type of testing is often referred to as black box testing where the result is binary (pass/fail) and cannot be audited. It should be noted that even in the case where a given test returns a spectrum of results it may still be possible to convert that test to a pass/fail paradigm. This may be achieved by having a threshold value and if the test returns a result over or under the specified threshold then a fail is returned.

How do we distinguish between passing and failing tests mathematically? We test for interactions, specifically failing interactions.

Definition 1.1.2. [6] Given a $\boldsymbol{TP(k; g_1, g_2, \ldots, g_k)}$ a **t-way interaction** is a t-tuple with each vertex in the tuple belonging to a different factor. If a t-way interaction T is a **failing interaction**, then every test (k-tuple) containing T will fail.

It should be noted that we assume implicitly that non-failing interactions vastly outnumber the failing ones for a given $\boldsymbol{TP(k; g_1, g_2, \ldots, g_k)}$, since if

this were not the case, it would be difficult to create any passing tests whatsoever. This is problematic since it is necessary to create a large number of passing rows to determine which interactions will fail. Having characterized some basic terms that we will need, we now move on to an example which will illuminate some of the problems encountered when studying the testing problem.

1.1.1 The Pizza Problem

Consider that we want to make a tasty pizza and while no ingredients on their own are going to make it taste bad, it is possible that some pair of ingredients will make it taste bad. We also assume we will pick one meat, one vegetable, one type of cheese and one type of sauce. We can represent these choices in a table (Table 1.1).

Table 1.1: The Pizza Problem

Sauce	**Vegetable**	**Cheese**	**Meat**
BBQ	Mushroom	Mozzarella	Pepperoni
Tomato	Peppers	Cheddar	Chicken
Italian	Onion	3-Cheese	Bacon

One critical assumption made regarding the testing problem is that we will not be able to ascertain which combination of ingredients is causing the pizza to taste bad. That is if a pizza tastes bad we will not know which combination of ingredients is causing the problem. This situation does arise

frequently in real world tesitng scenarios regarding food products [16]. With that in mind, we will actually create three different pizzas and then taste them to see if any ingredients are causing problems.

Table 1.2: The Pizza Problem Test Outcomes

Test	**Sauce**	**Vegetable**	**Cheese**	**Meat**	**How does it taste**
1	BBQ	Peppers	Cheddar	Bacon	Bad
2	Italian	Onion	Cheddar	Bacon	Good
3	Tomato	Mushroom	Mozzarella	Chicken	Good

Now that we have our results we wish to find the failing interactions in our ingredients. First we can immediately guarantee that any interactions in a pizza that tastes good will not be failing interactions, since if any were present the pizza would taste bad. We can further investigate which interactions are the failing interactions causing pizza 1 to fail. We can immediately rule out the cheddar cheese and the bacon, since this combination appears in a pizza which tastes good on row 2. In fact, we can eliminate all pairs on rows 2 and 3. However, this leaves 5 other possible pairs of ingredients which do not appear in either of the other two pizzas. So while our test found at least one failing interaction, it failed to locate it. We also remain unsure about the exact number of errors, it is possible that multiple bad interactions exist in pizza 1. Our array also did not cover all pairs, so it is possible that other failing interactions exist as well which have not been included in any of our 3 pizzas.

It is possible to cover all pairs of ingredients in 12 pizzas as shown in

Table 1.3. If all of these pizzas were to taste good we would be certain that no pair of ingredients would make any pizza taste bad.

Table 1.3: The Pizza Covering Array

Sauce	**Vegetable**	**Cheese**	**Meat**
BBQ	Mushroom	Cheddar	Bacon
BBQ	Peppers	Cheddar	Pepperoni
BBQ	Onion	Mozzarella	Pepperoni
Tomato	Mushroom	3-Cheese	Pepperoni
Tomato	Peppers	Mozzarella	Chicken
Tomato	Onion	Cheddar	Bacon
Italian	Mushroom	Mozzarella	Chicken
Italian	Peppers	3-Cheese	Bacon
Italian	Onion	Cheddar	Chicken
BBQ	Onion	3-Cheese	Chicken
Italian	Onion	Mozzarella	Pepperoni
BBQ	Onion	Mozzarella	Bacon

It should be noted that, this table will not tell us *which* pair of ingredients in pizza 1 from Table 1.2 causes the pizza to taste bad. If we presuppose the failing interaction is BBQ sauce and Mushrooms, this array will locate that failing interaction. However, if we supposed the failing interaction is BBQ Sauce and Onions the array will not. If we specify the array carefully and give a predefined set of good/bad tasting pizzas, it should be possible to create an array which would locate all the failing interactions with only a few extra rows assuming there are not too many failing interactions.

In practice since pizzas are usually ordered a few at a time we could adaptively order pizzas with one possible bad ingredient pair and all other

ingredient pairs being known "good" pairs, but as we shall see this is not always an option.

1.1.2 Practical Considerations

Ultimately, when we analyze a testing problem we are seeking failing interactions. Optimally, we would like to know not only how many failing interactions exist, but also which components they contain. Given the time and expense involved in running quality assurance tests, the interest from the private sector in this problem has been considerable.

A first effort might be to run all possible tests in a given TP. Since any TP implicitly contains a finite number of factors and within those factors a finite number of levels, it is possible to create a finite test suite covering every combination of components. The difficulty inherent to this method lies in the fact that the number of tests will increase exponentially in both the number of factors and the size of the factors. Explicitly, the number of tests required is g^k. Even a $TP(4;4)$ would require 256 tests to ensure all combinations are represented in a test. In real world situations a test suite's cost to execute will depend largely on the number of tests in the suite. Therefore, running the minimal number of tests for a given testing problem is vitally important.

One might try a "probabilistic" approach, where a random subset of all tests are selected, with the expectation that with a high probability all or

most failing interactions will be covered. However, starting with the premise that all failing interactions are pairwise, ie. all failing interactions contain two components, the fraction of tests which include any given interaction is only $\frac{1}{g^2}$. Thus unless the probabilistic test suite is almost as big as a complete one there is a high probability that a failing interaction will not be covered by a test.

Failing this we might consider an "adaptive" model which initially runs tests randomly, while ensuring that no interactions in a test which passes are covered twice. Then after a test fails, all interactions in the failing test are re-tested with known "safe" interactions from passing tests. This will precisely locate which interactions are failing interactions. However, in many real world test scenarios, tests must be scheduled well in advance and cannot be altered "on the fly" [7]. Thus the adaptive method is often not feasible. In addition the detection of "safe" values can itself be problematic.

What is needed is a method for constructing test suites which are smaller, in that they contain fewer tests than an exhaustive one. This suite must be defined in advance and are not presumed to be adaptive. Inevitably, not every interaction can be included in this smaller test suite. However, if certain restrictions are placed on a given TP, such as how many failing interactions are present or the placement of the failing interactions it may be possible to locate those interactions. With this as a starting point we begin laying out the necessary theoretical foundations for the discussion of

this problem.

1.2 Graph Theory Notation

As was previously suggested at the testing problem can be thought of as a graph theory problem; for this thesis we will need to define some basic graph theory terms before advancing any further, though we assume the reader is familiar with the basic concepts of graph theory. We will be using the text by West [19] as a basis for our graph theoretic terminology.

1.2.1 Basic Terminology

A **Graph** is defined a set of vertices, **V**, with a binary relation, **E**, called edges on each pair of elements forming the graph $\boldsymbol{G} = \{V, E\}$. We define $V(G)$ to be the set of all vertices on the graph G. We additionally define $E(G)$ as the set of all edges on the graph. We denote an edge between vertices x and y as simply xy. If an edge connects a vertex to itself it is said to be a **loop**. If two vertices share an edge they are considered to be **adjacent**. If we allow for the same two vertices to share multiple edges, it is said that we allow for **parallel edges**. A graph with no loops and no parallel edges is called a **simple graph**. For the purposes of this thesis, readers should implicitly assume that all graphs are simple.

There is a well known function which we will need to explicitly select the vertices adjacent to a given vertex. We define the **open neighborhood** of a vertex x or $N(x)$ to be the set of all vertices which are adjacent to x but not including x itself. The **closed neighborhood** of x or $N[x]$ is the set of all vertices which are adjacent to x as well as x itself.

A vertex's **degree** denoted, $d(x)$, is defined as the number of edges attached to that vertex. We define the **maximum degree** of a graph Δ as being the maximum number of edges any vertex has incident to it in G. Mathematically this can be represented as $\Delta = \max_{x \in V(G)} d(x)$. A **regular graph** is a graph where all vertices have the same degree. In many cases it suffices to consider the regular graph of degree Δ. Although in general a graph may have an infinitely large edge set $|E(G)|$ or vertex set $|V(G)|$. We will only be considering cases where the edge set and vertex set are finite in size. Such graphs are called **finite** graphs. We also define a graph G known as an **empty graph** if $|V(G)| \neq 0$ and $E(G) = \emptyset$. An **isomorphism** from a simple graph G to a simple graph H is a bijection $f : V(G) \to V(H)$ such that $uv \in E(G)$ if and only if $f(u)f(v) \in E(H)$. We say G is isomorphic to H, denoted $G \cong H$ if there exists an isomorphism from G to H.

1.2.2 Subgraphs and Special Graphs

We define H as a **subgraph** of G if $E(H) \subseteq E(G)$ and $V(H) \subseteq V(G)$, further the endpoints of the edges of E(H) must lie in V(H), this is denoted as $H \subseteq G$. We call it a **proper subgraph** when either $E(H) \subset E(G)$ or $V(H) \subset V(G)$. We denote this as $H \subset G$. Given a set of vertices $S \subseteq V(G)$ the subgraph **induced** by, S, which we call H, is the subgraph of G with $V(H) = S$ and given $x, y \in S$ $xy \in E(H) \Leftrightarrow xy \in E(G)$. We denote the edge set of this graph as $E(S)$.

We also define the notion of a complement graph, which will become important later. The complement of a graph G, known as the **complement graph** G^*, is the graph where $V(G^*) = V(G)$ and $e \in E(\mathcal{G}) \Leftrightarrow e \notin E(\mathcal{G}^*)$. We also define a special class of graphs, complete graphs. We define the **complete graph** on n vertices to be the graph K_n on n vertices in which all pairs vertices are **adjacent**. When an induced subgraph of G is itself a complete graph it is typically referred to as a **clique**.

There is a special set on a graph that will be important to this thesis in upcoming chapters known as the independent set.

Definition 1.2.1. Given a graph G, we define an **Independent Set**, S, as a set of vertices in a graph $S \subseteq V(G)$, where no two vertices in S are adjacent. That is to say $E(S) = \emptyset$.

There is a similar concept for a set which is adjacent to every vertex in a given graph. A **Dominating Set** defined on a graph G as a set $S \subseteq V(G)$ where every vertex in $V(G) \setminus S$ is adjacent to a vertex in S.

We can also define a special type of graph in which edges cover more than two vertices.

Definition 1.2.2. We define a **hypergraph** $\mathcal{H}$ as a set of vertices $V(\mathcal{H})$ and a set of hyperedges $E(\mathcal{H})$, each hyperedge consists of two or more vertices. We call $\mathcal{H}$ to be a $\boldsymbol{t}$-uniform hypergraph if all edges have size t.

Unlike the edges of a graph which contain a pair of vertices, hyperedges of a hypergraph may contain an arbitrary set of vertices.

1.2.3 Multipartite Graphs

We will also introduce the concept of the multipartite graph, which will be critical to our future work. In addition to the conventional graph theory terms, we will need some terminology relating to the class of graphs known as multipartite graphs.

Definition 1.2.3. A **multipartite graph** is defined as any graph G whose vertices can be partitioned into k independent sets. A multipartite graph with k parts of sizes $g_1, \ldots, g_k$ is denoted as $G(g_1, \ldots, g_k)$. If $g_1 = \ldots = g_k = g$ we denote this as $G_k(g)$.

If the parts are of different sizes we implicitly order them so that $g_1 \leq g_2 \leq \ldots \leq g_k$. Within a multipartite graph we will label a vertex v_i as being in the i^{th} part. Typically, when we are examining testing problems, we will presume all graphs to be of the form $G_k(g)$ and they may simply be referred to as G. We define a ***k*-tuple**, $(v_1, \ldots, v_k)$, as a set vertices $v_i \in G(g_1, \ldots, g_k)$, where one vertex from each part is present. We can extend this notion to the **complete multipartite** graph. We define a **complete multipartite** graph to be a graph $G(g_1, \ldots, g_k)$ in which $\forall x, y \in V(G)$, $xy \in E(G) \iff x$ and y are **not** in the same part. We denote such a graph as $K(g_1, \ldots, g_k)$.

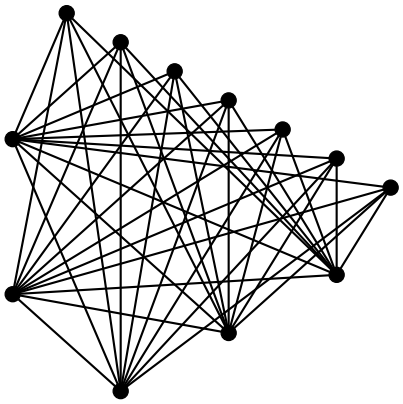

Figure 1.1: A complete multipartite graph $K(7, 3, 2)$

In Chapter 3 we will need the concept of the blow up graph, which should not be conflated with the better known blow up lemma.

Definition 1.2.4. Given a graph G of order k, we say that the k-partite graph $G'(g_1, \ldots, g_k)$ is a **blow up graph** of G, if there are edges between vertices v_i and u_j, $i \neq j$ of $G'(g_1, \ldots, g_k)$ (i.e. $v_i u_j \in E(G')$), only if there are

edges between the corresponding two vertices in G.

We also define the **complete blow up graph** of G to be the blow up graph where two parts of G'_k form a complete bipartite subgraph if and only if there is a corresponding edge in G.

We introduce a special version of the neighborhood function, which not only selects adjacent vertices but also those in a given part.

Definition 1.2.5. Given a multipartite graph $G(g_1, g_2, \ldots, g_k)$ and a point $x \in V(G)$ we define the function $\boldsymbol{M(x)}$, where $M(x) = N(x) \cup P(x)$ where $P(x)$ are the vertices of the part containing x. This includes the set of all vertices adjacent to x as well as all vertices in the part which contain x. We call $M(x)$ the **meighborhood** of x.

As an example the meighbourhood, of every point in a complete multipartite graph is every vertex in the graph.

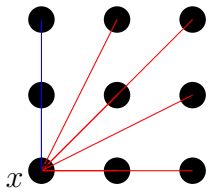

Figure 1.2: The Meighbourhood of x in a complete multipartite graph

Some special varieties of subgraphs will be useful in our discussion. We begin by defining a subgraph on the edge set of a set a graph G

Definition 1.2.6. Given a graph G, we define a graph H as an **edge subgraph** of G, when $V(H) = V(G)$ and $E(H) \subseteq E(G)$.

Important to this thesis is a particular type of subgraph which is unique to multipartite graphs, the partition subgraph.

Definition 1.2.7. Given a multipartite graph $G(g_1, \ldots, g_k)$, we define a **partition subgraph** $H(g_1, \ldots, g_l)$ to be an induced subgraph of $G(g_1, \ldots, g_k)$ obtained by removing entire parts from G. ie. $V(H) = \{v_j \in V(G) \mid j \in \{\sigma(1), \sigma(2), \ldots, \sigma(l)\}\}$, where σ is a permutation on $\{1, \ldots, k\}$ and $E(H) = \{v_i v_j \in E(G) \mid v_i, v_j \in V(H)\}$

When we discuss the partition subgraph we will typically discuss it in relation to a multipartite graph $G_k(g)$. We will refer to the partition subgraph as $H_l(g)$ where $l \leq k$, this will typically be done after removing the meighbourhood $M(a)$, where $a \in V(G_k(g))$, from $G_k(g)$ such that we are left with $H(r_1, \ldots, r_l) = G_k(g) \setminus M(a)$.

1.2.4 Turán's Theorem

We will find the well known Turán's theorem useful when discussing multipartite graphs. This theorem was proved by Hungarian mathematician Paul Turán. It has many implications for the construction of multipartite graphs. The construction of cliques on a multipartite graph is the opposite of

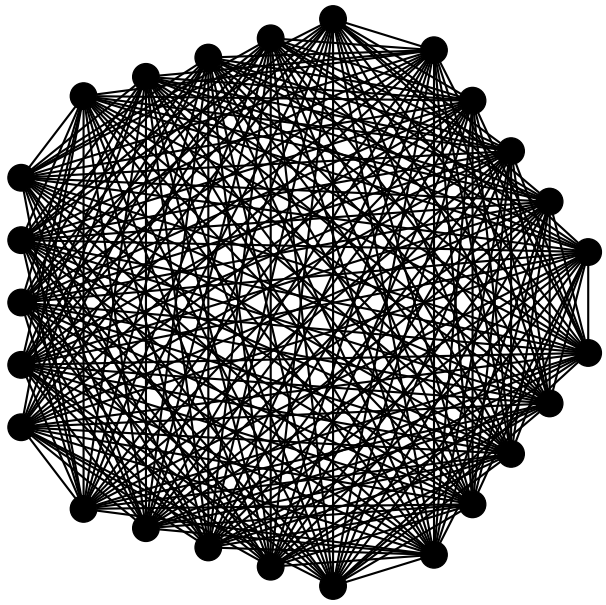

Figure 1.3: The Turan Graph $T_{25,5} = K(5, 5, 5, 5, 5)$

the construction of independent sets on that graph, so the utility of Turán's theorem is quite obvious. We begin by defining the Turán graph.

Definition 1.2.8. A **Turán graph** $T_{n,r}$, is defined to be a complete multipartite graph with n vertices over r parts, with each part differing by at most 1. By the pigeonhole principle, the smallest part has size $\lfloor n/r \rfloor$ and the largest has size $\lceil n/r \rceil$.

The accompanying theorem will allow us to construct our independent k-tuples later.

Theorem 1.2.9 (Turáns Theorem)**.** *[19] Among the n-vertex simple graphs with no $r + 1$ clique, $T_{n,r}$ has the maximum number of edges.*

The proof of this theorem is found in most graph theory textbooks, see [19] for example. However, we will mention that $T_{n,r}$ has at most $\left(1-\frac{1}{r}\right)\frac{n^2}{2}$ edges.

1.2.5 Computational Complexity and Decision problems

We assume that the reader is familiar with the basic concepts of complexity theory, for more information see [1]. In particular, a **decision problem** is a set within a known universe. A particular instance of the problem is in the set if it satisfies a given property. We will generally be operating on the set of graphs, or the set of multipartite graphs so we restrict our attention to this case. Generally an instance of a problem consists of a member of the universe, G (in our case a graph) and it is required to determine if G is in the set or not.

The question of complexity relates to the time an algorithm will take solve a problem. Every decision problem with a finite number of inputs can be solved eventually by simply exhausting all possible combinations of inputs. However, this is obviously not efficient and would require a lot of time, but there are obviously algorithms which can solve some problems more efficiently. Every algorithm we have developed in this thesis will run in linear time relative to the number of parts k on a given multipartite graph

$G(g_1, g_2, \ldots, g_k)$.

1.3 The Covering Problem

As previously mentioned, the total number of possible tests in a given TP will increase exponentially, in both k and g. This implies that the time required to actually run all these tests will also increase exponentially. However, it is possible to create an array which contains every possible t-tuple at least once without testing every possible k-tuple. Obviously, if $k = t$ we will need to exhaustively test the entire TP. However, when $t < k$ we create an array which is smaller than the exhaustive one. This allows for fewer total tests to be run for a given TP. In this section we will discuss this problem further.

1.3.1 The Error Graph

It is possible to define the interactions of any two components in a testing problem where we presume $t = 2$ as a graph G. This graph is necessarily multipartite with k parts. Any adverse interactions between two components can be represented as an edge between the two corresponding vertices. We wish to determine when such an edge exists.

Before we can begin our discussion we must consider what a passing test

would look like in a graph theory context. Here a passing test is labelled a passing row.

Definition 1.3.1. Given a multipartite graph $G(g_1, g_2, \ldots, g_k)$, take a k-tuple, S, containing a given pair of vertices $v_i, v_j \in V(G)$, with $i \neq j$. A **passing row** for v_i, v_j is a row, (k-tuple) S which is independent in G and a failing row if S is not.

With this simple transformation we can take a testing problem and convert it to a graph theory one. Specifically a design problem where we attempt to take two vertices from two separate parts of a multipartite graph and try and construct an independent k-tuple on the remaining partite subgraph $H_{k-2}(g)$. While most of the interactions we deal with in this thesis are pairwise, and so are represented by graphs, it is straightforward to generalize to the case of arbitrary t-wise interactions, which are represented by hypergraphs.

We have previously defined a t-way interaction in Section 1.1 as a t-tuple, we now begin to look at the testing problem not as a combinatoral object but as a graph. In most cases we will be implicitly considering only the case where $t = 2$ and therefore we can limit ourselves to conventional graphs as opposed to hypergraphs. We begin by introducing the notion of avoiding a given graph G

Definition 1.3.2. We say that a set of vertices S **avoids** a graph G if no

pair of vertices $v_i, v_j \in S$ is an edge of G.

If S includes one vertex from each part of a multipartite graph $G_k(g)$, this set not only avoids $G_k(g)$ but would constitute a passing row as specified in Definition 1.3.1. This definition will be critical to the construction of locating rows in Chapters 2 and 3.

1.3.2 Covering Arrays

The covering problem has been considered extensively [2, 3, 4, 6, 8, 9, 12, 13, 15, 18]. The problem has a particularly large range of applications when one is considering all the possible t-way interactions between software configurations. Studies have shown [10] that most configuration problems in a software system are between two settings, so the pairwise interaction remains the most commonly studied model. Covering problems have attracted attention from mathematicians as well as computer scientists. Due to this parallel development, there are often divergent terminologies for identical concepts. We will adopt various terminologies as they seem appropriate.

As we have previously mentioned, exhaustively testing all possible combinations in a testing problem is prohibitively expensive and often unecessary [10]. Given these limitations the problem then becomes one of actually designing a test suite which tests some number of pairs but not all. Extenisve work [2, 3, 4, 6, 8, 9, 12, 13, 15, 18] has already been done the problem of

covering arrays, giving us a good initial definition to work from.

Definition 1.3.3. [3] A **Covering Array** denoted as $CA_\lambda(N, t, k, g)$, is an $N \times k$ array A, with entries from an alphabet of size g. We have all t-tuples taken across all parts appearing at least λ times over all rows in A. N is the **size** of the array, here meaning the number of rows, t is the **strength** of the interaction that we are testing; k is the number of factors; g is the size of the alphabet, called the **order**. λ is known as the **index** and typically is set as 1 and is omitted.

We generate an array which covers a given TP. For example, for a $TP(4, 2)$ we can construct an array which covers all pairs as show below in Table 1.4 [5].

If any interactions are present at least one row will fail. However, if any rows given above fail we will not know which interaction caused the failure. Although there are many possible arrays that would cover a given $G(g_1, \ldots, g_k)$, there must be some, smallest array for a given $CA(N, t, k, g)$.

Definition 1.3.4. We define the **covering array number** or **CAN** as the minimum N for which a $CA(N, t, k, g)$ exists.

We also note that the smallest CA need not be unique, there may be many possible minimal CA's. The size of the covering array has been determined asymptotically for strength $t = 2$.

Table 1.4: Covering array ($t = 2$ for $k = 4$, $g = 2$)

0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

Theorem 1.3.5. *[8] Let $g \geq 2$ be fixed, Then as $k \to \infty$, the upper limit on the covering array number behaves as follows.*

$$CAN(2, k, g) \sim \frac{g}{2} \log_2 k$$

It interesting to note that while exhaustive testing will increase exponentially in both k and g, when we use covering arrays, the increase is linear in g and logarithmic in k.

We can also consider the case where not all parts of a graph are of equal size that is we allow for a graph $G(g_1, g_2, \ldots, g_k)$ we allow for one or more $g_i \neq g_j$, for this case we obtained a mixed covering array.

Definition 1.3.6. [6] A **Mixed Covering Array** denoted as $MCA_\lambda(N, t, g_1, \ldots, g_k)$, is an $N \times k$ array A, with the i^{th} part containing an alphabet of size g_i. We have all t-tuples accross all parts appearing at least λ times over all rows in A. As before N is the **size** of the array, here meaning the number of rows, t is the **strength** of the interaction that we are testing; the set of values $\{g_1, \ldots, g_k\}$ are size of the alphabets in each part of the array, called the **order**. λ is known as the **index** and typically is set as 1 and is omitted.

The MCA has an a analogue to the covering array number

Definition 1.3.7. For a given t, k, g has a **mixed covering array number** or **MCAN**$(t, (g_1, \ldots, g_k))$, is the smallest N for which an MCA$(N, t, (g_1, \ldots, g_k))$ exists.

Definition 1.3.8. [6] Given a multipartite graph $G_k(g)$ We define a **covering array avoiding forbidden edges** or **CAFE**. This is defined as an $N \times k$ array A, with each column i having symbols from an alphabet of size g, such that each row in A forms a k-tuple which avoids all edges in G and between all rows all pairs of vertices which avoid $G_k(g)$ are covered.

Much like covering arrays and mixed covering arrays we can define a CAFE number. We denote $CAFEN(G)$ as the minimum N for which a $CAFE(N, G)$ exists.

1.3.3 The Decision Problems COVER and AVOID

We define some decision problems relevant to our work. These results come from [6].

Definition 1.3.9. A graph $G(g_1, \ldots, g_k)$ is said to be in **AVOID** if there is a k-tuple denoted as S where $\forall v_i, v_j \in S, v_i v_j \notin E(G)$.

We can also state that if a graph G is in AVOID then every pair of vertices in G can be put into a passing row.

Definition 1.3.10. A graph $G(g_1, \ldots, g_k)$ is in **COVER & AVOID** if for every t-tuple, T, there is a k-tuple S, such that S contains T and S avoids G', where $G' \cong G(g_1, \ldots, g_k) \setminus T$.

We can extend the definition of AVOID to include hyperedges of size t, by introducing the concept of AVOID(t).

Definition 1.3.11. A graph $G(g_1, \ldots, g_k)$ is in **AVOID(t)** if and only if given any t-tuple T, $\exists$ a k-tuple S, where $T \subseteq S$, such that S avoids G' where $G' \cong G(g_1, \ldots, g_k) \setminus T$, ie. S is a locating row in G'.

It is important to note that in general determining whether or not a graph G is in AVOID or AVOID(t) is an NP-complete problem [6]. These definitions come from the notion of the covering array as discussed in Section 1.3.2.

1.4 Locating

Obviously, simply covering all interactions will not necessarily be enough to actually locate all failing interactions. For example, if we presume that we are looking for pairwise failing interactions and our covering array covers each pair exactly once then any failing row will contain a possible $\binom{k}{2}$ interactions which caused the failure. As a further confounding factor, if a given row contains multiple failing interactions the pass/fail nature of the testing we conduct does not yield any information about the number of errors in a given row. This was briefly intimated in Section 1.1.1 but now we will discuss this problem in more explicit detail.

If we are interested in a known subset of interactions, denoted as I, then we would like to construct an array which will locate those errors, if they are present. In a graph theory context, a given TP has associated with it a family of graphs $\mathcal{G}$ which contains all possible combinations of failing interactions, represented as edges. However, there is one graph $G \in \mathcal{G}$ which represents the *actual* configuration of failing interactions that a given TP has. With this in mind, we can think of a locating array as a kind of meta-test where we presume a given G and the locating array will return either a pass if that G is the actual error graph corresponding to a given TP or a fail if it is not. In order to do this, we must first define what it means, in general, for an interaction as well as a hypergraph to be locatable.

1.4.1 Locatability

It is not always possible to create a locating row containing any two non-adjacent components for all possible $G_k(g)$. Indeed, an example is given in Sections 1.4.5 and 1.4.6 many more examples will be given in Chapter 2 of non-locatable graphs, where there exsists least one interaction which can not be located. Locatability is a major concern when conducting tests. It is good to know that errors exist but it is important to know which interactions are causing them. As we shall see in the next chapter it is possible for two or more interactions to "mask" another. There are a few ways to define locatability. If it is possible, for a given G, to construct an $ELA(n, G)$, defined in Section 1.4.2, where $n \in \mathbb{Z}$ then the graph G is **locatable**. However, this is not a particularly informative definition and it would be good to specify exactly under what circumstances a graph is or is not locatable. While there are other equivalent definitions [15] we use the following.

Definition 1.4.1. A t-way interaction T, is **locatable** if and only if it is possible to create a k-tuple, S, where $T \subseteq S$, and $E_{G'}(S) = \emptyset$, where $E(G') = E(G)\backslash\{T\}$ and $V(G') = V(G)$. We say S **locates** T and we call S a **locating row** for T.

A t-uniform hypergraph G is t-locatable if and only if all interactions $T \in E(G)$ are locatable.

For this thesis, we presume that $t = 2$. Therefore, we can restrict ourselves

to considering conventional edges as opposed to more general hyperedges. In this context, a graph is locatable if and only if every pair of vertices in G can be placed in a locating row, see Definition 1.4.1. It should be noted that there is a connection between a graph G being locatable and being in AVOID(2).

Lemma 1.4.2. *If a graph G is locatable then G is also AVOID(2).*

Proof. The proof comes from the definition of AVOID(2), if we make the xy our t-tuple, then by the definition of locatablility, it must be possible for that tuple to avoid G. □

1.4.2 Error Locating Arrays

Martínez, Moura, Panario, and Stevens [15] define an Error Locating Array as an array which both locates and detects errors for a graph which satisfies certain conditions. In Section 1.5, we will further discuss the sometimes counter intuitive distinction between locating and detecting errors. However, the definition of the Error Locating Array ignores such distinctions.

Definition 1.4.3. An **Error Locating Array** or **ELA** for a graph $G(g_1, \ldots, g_k)$, denoted by ELA(N, G), is an $N \times k$ (mixed) covering array A with each column i having symbols from an alphabet of size g_i, such that every interaction $\{v_i, v_j\}$ corresponding to a pair of vertices $v_i, v_j \in V(G)$ with $i \neq j$ is located by a row of A.

It should be noted that there is no requirement that an ELA be a minimal covering array, it may take many "additional" rows to create an array that has the property we are looking for. It should be noted briefly that a graph has an ELA if and only if it is locatable. For an example of a non-locatable graph see section 1.4.5.

The construction of an Error Locating Array is in general an NP-complete problem [6, 13]. In fact even constructing a locating row is in general an NP-complete problem.

Theorem 1.4.4 (locating row complexity)**.** *[6] Constructing a k-tuple S which avoids G is an NP-complete problem.*

The proof of this theorem requires some descion theory concepts not developed in this thesis and is covered in Danziger, Mendelsohn, Moura and Stevens [6].

1.4.3 Conditions for Locatability

The total number of interactions that given graph $G(g_1, g_2, \ldots, g_k)$ may have has been determined [15] as being one less than the size of the smallest part g_1. However, this is a coarse measure and simply having g_1 edges does not guarantee that a given graph $G(g_1, g_2, \ldots, g_k)$ is non-locatable. Particular edge configurations are required to actually make a graph non-locatable.

1.4.4 Safe Values

As previously mentioned not all graphs are locatable. However, if a G has certain properties, it is possible to guarantee locatability. In Martínez, Moura, Panario, and Stevens [15] the concept of "safe values" is discussed, safe values occur when we know certain vertices to be **safe**. That is, a vertex $s_1 \in V(G)$, is safe if for every $v_i \in V(G)$, is no $s_1 v_i \notin E(G)$. Now consider, if we have at least one safe vertex in every part of a given $G(g_1, \ldots, g_k)$, then $G(g_1, \ldots, g_k)$ must be locatable, since we can construct an ELA where k-tuple would contain $\{x, s_1, \ldots, s_{k-2}, y\}$ and if the row failed the failing interaction would be immediately identified as xy. We can also introduce the notion of a **safe set** denoted by S_j which is a set of j safe values from j different factors such that $S_j = \{s_1, s_2, \ldots, s_j\}$.

1.4.5 An example of a non-locatable tripartite graph

When discussing locatability it is instructive to consider a case where a graph will not be locatable. Consider the following graph $G_3(g)$, leaving g arbitrary. Take two vertices from two distinct parts, x and y. Label all the vertices in the remaining part v_i. If for every v_i, either $xv_i \in E(G)$ or $yv_i \in E(G)$ then G is non-locatable. Note that if any $G_k(g)$ has this configuration as an partition-subgraph then that $G_k(g)$ is non-locatable.

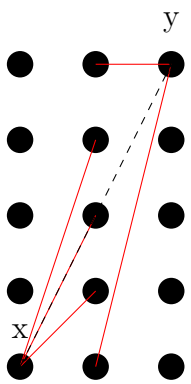

Figure 1.4: A non-locatable graph $g_3(5)$

Now it is not possible in any k-tuple containing x and y to determine whether $xy \in E(G)$.

Martínez, Moura, Panario, and Stevens [15] describes a theorem which determines a condition on whether a graph is non-locatable.

Theorem 1.4.5 (Locatablility)**.** *[15] Let $G = G(g_1, g_2, \ldots, g_k)$, with $k \geq 3$*

1. *If there exists $v_i \in V(G)$ and a factor $j \in [1,k] \setminus \{i\}$ such that $\{v_i, v_j\} \in E(G)$ for all $v \in V(v_j)$ then G is not locatable.*

2. *If there exists $v_i, v_s \in V(G)$ with $i \neq s$ and a factor $j \in [1,k] \setminus \{i,j\}$ such that for all $v \in V(v_j), \{v_i, v_j\} \in E(G)$ or $\{v_s, v_j\} \in E(G)$, then G is not locatable.*

Proof. It is enough to prove the second statement, since the first is a special case of the second, by taking $s \in [1,k] \setminus \{i,j\}$. Let $G' = G \setminus \{v_i, v_s\}$ if

$\{v_i, v_s\} \in E(G)$, and $G' = G$ otherwise. It is easy to see that there is no k-tuple, T avoiding G' since either some $\{v_i, v_j\} \in T$ or $\{v_s, v_j\} \in T$ covers an edge of G'. □

It should be noted that the degree conditions which allow for Theorem 1.4.5 to be relevant are not discussed here and will be discussed in Section 2.3.1.

1.4.6 Non-Locatability for binary alphabets $g = 2$

If we are examining an error graph with at least 3 parts of size 2, then all possible non-locatable graphs on those parts have been characterized by Martínez, Moura, Panario, and Stevens [15]. In addition, the NP-Completeness of locating an error in such a graph has been determined by Maltias [11].

In general, there are 6 non-locatable subgraphs in the "binary", $g_i = 2$ case. However, there are 2 possible non-locatable graphs for $g = 2$ and $t = 2$ [15]. If a graph $G_k(2)$ contains at least one of the following graphs in Figure 1.5 as a partition subgraph, it will not be possible to locate the edge shown in the dashed line.

Figure 1.5: Non-Locatable binary graphs

1.5 Locating and Detecting Arrays

Confusingly, two different formulations have been developed; a theoretical approach [15] and a practical design approach [3]. While we focus on the graph theory approach from [15], there are equivalent formulations. We will engage in a brief discussion of the work done by Colbourn and McClary in characterizing locating and detecting arrays [3]. We will have to characterize some terms before moving forward. In Colbourn and McClary arrays are characterized by the strength of the interaction t and the total number of interactions d. These are appended by either the term locating or detecting depending on what type of array we are describing, so arrays will be either (d, t)-locating or (d, t)-detecting. The symbols $\bar{t}$ and $\bar{d}$ denote arrays that will either detect or locate, up to t strength interactions or number of interactions respectively. It should be noted that we have not discussed which errors actually exist in a given TP and what their strengths are. We are only interested in detecting errors in a given array. With the terminology specified, we can now begin discussing the actual results of Colbourn and McClary.

1.5.1 Locating Arrays

Now that we have described the concept of the locating array we can now provide a more technical definition of what exactly constitutes a locating array. In contrast to an Error Locating Array, Colbourn and McClary [3] characterizes a locating array as an array where if a set of errors are present they will be located by the array. Also, in locating arrays, only the number of failing interactions d and the strength of those interactions t are considered, as opposed to Error Locating Arrays where the nature of the error graph is taken into account.

Definition 1.5.1. Given a $TP(k; g_1, \ldots, g_k)$ and a specified set of t-way interactions I, where $|I| = d$ and all failing interactions are presumed to be in I. A Locating array LA(d, t) is an $N \times k$ covering array $MCA(N, t, g_1, \ldots, g_k)$ such that the set of Pass/Fail results obtained by running the N tests of the MCA are unique for each configuration of failing interactions in I.

It should be noted that sometimes an array will be said to be (d, t)-locating rather than a $LA(d, t)$. It should be noted that a LA must also be a covering array, although it may not be a minimal one. It should also be noted that if it is possible to construct a LA for a given graph G then that graph is **locatable**. An example of a LA(1,1) is given below in 1.5 [3].

Table 1.5: Locating array $t = 1$ and $k = 6$, $g = 3$

0	0	0	0	0	0
0	0	1	2	2	1
0	1	0	1	2	2
1	2	1	0	1	2
1	2	2	1	0	1
1	1	2	2	1	0

Interestingly, the lower bound on the number of rows any locating array will have has not been determined. In the case where exactly one failing interaction is to be located, the lower bound has been established [17]. However, while arrays which locate multiple errors have been shown to exist, the minimum bounds on such arrays are an open problem. It should also be noted that the actual construction of locating arrays is in general NP-Complete [6].

1.5.2 Detecting Array

As previously discussed, a locating array will not determine which interactions are present. However, if we presuppose a certain number of interactions in a given testing problem, then there will be a finite number of configurations that those interactions may fall into. It is obvious that if it is possible to construct an array which will locate a given set of interactions it must be possible to construct an array which will detect all such interactions. This is known as a detecting array.

Definition 1.5.2. [3] An array $A(d, t)$ is said to **detect** up to d faults of

strength t if the set of all interactions I can be determined from the test outcomes. This is denoted as $DA(d, t)$.

A simple way to think about $DA(d, t)$ construction is to imagine creating an $LA(d, t)$ for each combination of faults up to d and running them in sequence, removing any identical rows. While locating arrays will only locate one graph G detecting arrays will locate every graph in $\mathcal{G}$, since a detecting array is constructed from multiple locating arrays constructing a minimal error detecting array will be an NP-complete problem.

1.5.3 Conditions for Locating and Detecting Arrays

We explore the relationship between the number of errors d, in a given TP and the strength of the interactions t.

Lemma 1.5.3. *[3] If $t = k$ then the exhaustive array E, is a $(\bar{d}, t)$-detecting array for all d.*

This property emerges from the definition of locatability. If the strength of the interactions is exactly the size of a row, then any failing interactions must be the only interaction in that row.

Lemma 1.5.4. *[3] A $(d, \bar{t})$-detecting array is a $(\bar{d}, \bar{t})$-locating array and a $(d, \bar{t})$-locating array.*

$(d, \bar{t})$-detecting $\rightarrow$ $(\bar{d}, \bar{t})$-detecting $\rightarrow$ $(\bar{d}, \bar{t})$-locating

Figure 1.6: Relation between Detecting and Locating Arrays

This property simply emerges from the fact that if an array *detects* a set of interactions it must have *located* that set to do so, therefore any detecting array is a locating array.

We can visualize the relationship between detecting arrays (DAs) and locating arrays (LAs) in a helpful flowchart shown below. the terminology here is particularly unfortunate since, one might imagine that a locating array would determine where, in a given TP faulty interactions would actually exist and a detecting array would "detect" if a given set of interactions is present. It should also be noted that an array is an Error Locating Array if and only if it is a $(\bar{d}, \bar{t})$-detecting array [15].

1.5.4 A theorem for mixed strength loctability

Although we focus on regular graphs with pairwise edges, the concept of locatability can apply to hypergraphs in general. In particular, it should be noted that if a hypergraph is t locatable it is locatable up to t as well.

Theorem 1.5.5 ($\bar{t}$-locatability)**.** *If a hypergraph $H(g_1, \ldots, g_k)$ with fixed degree Δ is in avoid(t), then it is also in avoid($\bar{t}$), where $\bar{t} \leq t$.*

Proof. Any vertex x in $H(g_1, \ldots, g_k)$ must be contained in only Δ failing

interactions, T-tuples, of size t. Since it is possible to put any $T \subseteq S$, such that S avoids $H' \cong H(g_1, \ldots, g_k) \setminus T$, then it must be possible to put $\bar{T} \subseteq T$, where $\bar{T}$ is of size $\bar{t}$ into that same S and it will avoid $H' \cong H(g_1, \ldots, g_k) \setminus \bar{T}$ □

While we do not examine cases other than $t = 2$ this result is still important to the construction of Error Locating Arrays.

1.6 Overview of Thesis

We have already established the necessary concepts that we need to discuss the topics covered later in this thesis. We will exclusively focus on degree constraints that will guarantee whether or not a graph is locatable. We will not concern ourselves with constructing detecting arrays which is in general an NP-Hard problem [15]. In this thesis we focus on the error graph approach as opposed to looking at conditions on locating arrays. While the general size an error graph can have while still being locatable has been well described, the question of whether of not restricting the degreee of an error graph, so that we can guarantee it's locatability, is the subject of this thesis.

In Chapter 2, we will discuss the problem of locatability and examine particular cases in which a given error graph will be non-locatable. We will also look at some examples where a given graph will certainly be locatable if

certain degree constraints are satisfied. In Chapter 3, we will discuss specific constraints on the **degree** of a given graph, which guarantees the locatability of a graph, in this case for arbitrary values of k. In Chapter 4, we will review our conclusions and discuss some open problems which remain.

Chapter 2

Special Cases of Locatability

We are now ready to look at some special classes of graphs in which either locatability can be inferred from a given degree property of the graph or they are non locatable. First we will need to establish some basic theorems that will allow us to prove the locatability of our subsequent graphs. In order to proceed, we need to know that we can delete edges from a graph whilst maintaining locatability. In section 2.3 we touch upon some special cases where a given graph $G(g_1, g_2, \ldots, g_k)$ is not locatable. These cases require specific configurations which would necessitate that certain vertices have a particular degree. This leaves open the question:

What if we restrict the degree of all vertices on a given graph $G(g_1, g_2, \ldots, g_k)$*? Can we then guarantee that the graph is locatable?*

In some special cases the answer is reasonably straightforward, particularly when we can restrict the number of parts k. We will examine these special cases in this section.

2.1 Subgraph Locatability

We begin by considering cases when locatability is inherited by a subgraph. We would like a theorem which guarantees that edge deletion can not make an otherwise locatable graph non-locatable.

Theorem 2.1.1 (Locatability of Edge Subgraphs)**.** *If a graph G is locatable, then any edge-subgraph, H, of G is also locatable.*

Proof. Consider that if G is locatable then $\forall x, y \in V(G)$ there is a locating row S, where $S \subseteq V(G)$ with $E_{G'}(S) = \emptyset$, and $G' = G \setminus \{xy\}$ where $E_{G'}(S)$ is the edge set of the graph induced by S in G'. Then on any subgraph H where $V(G) = V(H)$ and $E(H) \subseteq E(G)$, S will still be a locating row and therefore H is locatable. □

This Theorem means that it suffices to consider only multipartite regular graphs of degree Δ. Since if any regular graph with degree Δ is locatable, then a graph where Δ is the maximum degree of any vertex is also locatable.

We can generate an almost identical theorem relating to the partition

subgraph. This will become important later in this section when we need to operate on particular partition subgraphs.

Theorem 2.1.2 (Locatability of Partition Subgraphs)**.** *If a graph $G_k(g)$ is locatable then any partition subgraph $H_l(g) \subseteq G_k(g)$ is also locatable.*

Proof. By assumption $H_l(g)$ is formed by deleting parts from $G_k(g)$. So any locating row in $G_k(g)$ contains an independent set in $H_l(g)$ which covers every part of $H_l(g)$. □

The contrapositive of Theorem 2.1.2 will also be useful. If there is some partition subgraph $H_l(g) \subseteq G_k(g)$ which is non-locatable, then $G_k(g)$ is non-locatable. This leads to an interesting corollary.

Corollary 2.1.3. *A graph $G_k(g)$ is locatable if and only if every partition subgraph $H_l(g) \subseteq G_k(g)$ is locatable.*

With these two theorems we can begin to generate some specific examples where a graph is locatable regardless of the actual edge configuration is, as well as show some examples where a graph contains a non-locatable edge configuration.

2.2 Safe Values

Recall that a vertex is called safe in a given graph $G_k(g)$ if that vertex has degree 0, i.e. is an isolated vertex. If some vertex, s_1, is known to be **safe**, it can immediately be put into a locating row with any vertices x and y, $\{x, s_1, \ldots, y\}$, without any further examination. Interestingly, any part of a graph $G_k(g)$ which contains a safe vertex can effectively be removed from the graph, unless one of the vertices we are testing is in that part. Additionally, we can create a **safe set** with safe values $S = \{s_1, s_2, \ldots, s_j\}$. The vertices in the safe set effectively remove the parts which contain them so that the the number of factors we need to consider becomes $k - j$. If there is at least one safe vertex in each part of a given graph $G_k(g)$ then that graph is always locatable.

2.3 Some Cases of a Non-Locatable Graph

We shall begin by generating some examples where a given graph is not locatable.

2.3.1 A non-locatable 3-Partite graph

As discussed in Section 1.4.5, it is possible to construct a graph which has at least one interaction which cannot be located by any 3-tuple. We give such a graph below where there exists a non-locatable edge between vertices x and y.

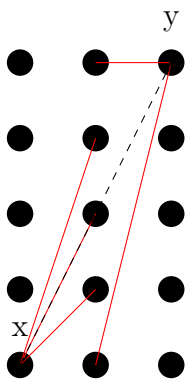

Figure 2.1: A non-locatable 3-partite graph

There is no way to put x and y into a locating row since any k-tuple containing x and y will also contain a separate failing interaction, therefore, the graph depicted in Figure 2.1 will not be locatable. It should be noted again that if some graph contains the graph referenced in Figure 2.1 as a partition subgraph, that graph will be non-locatable.

2.3.2 A non-locatable 4-Partite graph

In all cases that we have discussed so far, the two "test" vertices have been adjacent to all vertices in one part. So that after the neighbors of those two vertices are removed there exists a part of the graph with no vertices remaining. However, as we will see this does not necessarily have to be the case and it is possible remove the meighbourhood of the two "test" vertices $M(x)$ and $M(y)$, and still have live i.e. non-adjacent vertices in all remaining parts, but the graph is non-locatable.

We can formulate a non-locatable interaction in a 4-partite graph. We again construct a graph which contains an non-locatable interaction between vertices x and y.

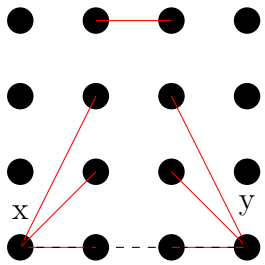

Figure 2.2: A non-locatable 4-partite graph

Clearly there is no row that locates x or y since the only vertices not adjacent to either of those vertices are adjacent to each other.

Interestingly, if any part of this graph is removed it will become locatable.

It is also interesting to note that this graph has vertices in every part which are not adjacent to x or y and yet is still not locatable. This graph therefore serves as an important example demonstrating that an interaction may only be non-locatable if the entire graph is taken into account, and not over some number of parts in the graph, locatability is a global . It also should be noted again that the graph shown in Figure 2.2 requires some vertices have a degree greater than $\frac{g-1}{2}$, to actually construct.

2.4 Locatability with $\Delta = 1$

We begin our discussion by looking at a simple case. What if for any graph $G(g_1, g_2, \ldots, g_k)$ all vertices are restricted to a degree of one? From section 1.4.6 we know that if we restrict the size of our parts to the binary case where $g_i = 2$, such a graph can be non-locatable. However, if we presume that $g_i \geq 3$ and a maximum degree $\Delta = 1$, then we will now show that a graph $G(g_1, g_2, \ldots, g_k)$ is locatable.

Theorem 2.4.1. *If a graph $G(g_1, g_2, \ldots, g_k)$ where all parts $g_i \geq 3$ and $\Delta = 1$, then $G(g_1, g_2, \ldots, g_k)$ is locatable.*

Proof. Starting with $G(g_1, g_2, \ldots, g_k)$, we must show that given any two vertices x and y, where $x, y \in V(G(g_1, g_2, \ldots, g_k))$, we can create a locating row containing x and y. If we delete $M(x)$ and $M(y)$, we are left with the

partition subgraph $H(r_1, r_2, \ldots, r_{k-2})$, where the part R_i with size r_i is the remaining vertices from g_i. We reorder the parts from smallest to largest so that r_1 is a part of smallest size and r_{k-2} is the largest, although there may be other parts equally as large. Since $\Delta = 1$, we removed at most 2 vertices and so $r_i \geq 1 \ \forall i$. Let $a_1 \in R_1$, if we delete $M(a_1)$ and reorder the new partition subgraph H, so that $r'_1 \leq \ldots \leq r'_{k-3}$, then the next smallest part is still of size $r'_1 \geq 1$. Again, every part in the remaining graph has size at least 2. Similarly we now select $a_2 \in R_2$ and delete $M(a_2)$. Again, after this step and subsequent reordering the smallest part is of size $r''_1 \geq 1$. We can continue this process sequentially with no part R_i ever being empty. Thus $G(g_1, g_2, \ldots, g_k)$ must be locatable. □

The proof of Theorem 2.4.1 implies an algorithm to directly construct a locating row on a graph $G(g_1, g_2, \ldots, g_k)$ with $\Delta = 1$ given any two x and y where $x, y \in V(G(g_1, g_2, \ldots, g_k))$.

Algorithm: 2.4 a $\Delta = 1$ locating Row Algorithm

Begin Algorithm

Function $\Delta = 1$ locate (G, k, x, y)

\\Inputs a multipartite graph G with k parts $R_i, \ldots, R_k$

\\and two vertices x and y in different parts of G

$S = \{x, y\}$

$G = G \setminus M(x)$

$G = G \setminus M(y)$

For $(i = 1$ **to** $k - 2)$

Sort$(R_1, \ldots, R_i)$ \\Here we are sorting parts from smallest to largest

If $|R_1| = 0$

Return *Fail*

$a =$ **Random Select**(R_1) \\Selects a random vertex from R_1

$S = S \cup \{a\}$

$G = G \setminus M(a)$

Next i

Return S

End Algorithm

It is clear that this algorithm will run for k steps, at which point a locating row, a k-tuple, will have been created. Thus algorithm 2.4 will execute in $\mathcal{O}(k)$ time. This algorithmic approach suggests that it might be possible to not only prove the existence of a locatable graph given certain constraints but that it is possible to create all the necessary locating rows in linear time. Although running algorithm 2.4 for every pair of vertices on a given $G(g_1, g_2, \ldots, g_k)$ will not necessarily create a minimal testing suite, it will locate all errors. However, even with this simple bound the number of allowable errors in a given TP has risen from $g-1$ [3] to gk, a significant increase, with the proviso that no single vertex is the cause of too many errors.

2.5 Errror Graphs with fixed part sizes

2.5.1 Special Note on $G_2(g)$

In case where a graph contains only 2 factors it is certainly pairwise locatable regardless of degree, since any two vertices x and y in $G_2(g)$ will form a k-tuple with at most one failing interaction. The interaction between x and y themselves. Thus every failing row definitively locates the pair it covers. Therefore any covering array is also a pairwise locating and detecting array if $k = 2$.

Interestingly, we can make a statement about 1-way interactions on $G_2(g)$. Specifically, if we restrict the degree of a $G_2(g)$ graph to $\Delta < g$ then it is locatable for $t \leq 2$ since if there exists at least one x and y in $G_2(g)$ which passes then neither x nor y contains a 1-way failing interaction. If for any x all rows fail then it must have contained a 1-way interaction. Otherwise there should have been at least one passing row containing x since $\Delta < g$.

2.5.2 A locatable graph $G_3(g)$

We have discussed in section 1.4.5 a special case where a graph $G_3(g)$ would be non-locatable. However, we can use Theorem 1.4.5 from [15] to create a degree bound which guarantees that a given graph $G_3(g)$ will be

locatable.

Theorem 2.5.1. *[15] If a graph $G_3(g)$ has a maximum degree $\Delta \leq \lfloor \frac{g-1}{2} \rfloor$ then it is locatable.*

Proof. Consider there are two vertices $\{x, y\}$. There is only one remaining factor R_1 after removing $M(x)$ and $M(y)$ this factor has at least $|R_1|-2\Delta \geq 1$ vertices. This remaining vertex can be put into a 3-tuple with $\{x, y\}$, to create a locating row on $G_3(g)$. □

It should be noted that adding only one more edge could remove the last vertex available to us and make part R_1 empty, which would in turn make the graph non-locatable. This method suggests that our algorithm can be adapted to this tripartite case.

Begin Algorithm

Algorithm: 2.5.2 a $G_3(g)$ locating Row Algorithm

Function $G_3(g)$ Locater (G, x, y)

\\Inputs a multipartite graph G with 3 parts and two vertices x and y

\\in different parts of G

$S = \{x, y\}$

$G = G \setminus M(x)$

$G = G \setminus M(y)$

If $|V(G)| = 0$ \\If the remaining graph G is empty the algorithm fails.

Return *Fail*

$a =$ **Random Select**(G) \\Selects a random vertex from the

\\remaining part

$S = S \cup a$

Return S

End Algorithm

2.5.3 A locatable graph $G_4(g)$

We have established that a tripartite graph can be locatable if we restrict $\Delta = \lfloor \frac{g-1}{2} \rfloor$. Can we say the same for a 4-paritite graph? As it turns out the same bounds can be applied to $G_4(g)$.

Theorem 2.5.2. *If a graph $G_4(g)$ has a maximum degree $\Delta \leq \lfloor \frac{g-1}{2} \rfloor$ then it is locatable.*

Proof. Consider two vertices $x, y \in V(G)$. We can then delete $M(x)$ and $M(y)$. Since there are 4 parts if there is at least one vertex remaining in each of the other parts of the graph which are non-adjacent then those 2 vertices can form a locating row with x and y. Consider that $M(x)$ can delete at most $\frac{g-1}{2}$ vertices from the remaining two parts, the same can be said of $M(y)$. Therefore collectively we can delete at most $g-1$ vertices from the remaining subgraph. The remaining partition subgraph $H_2(r_1, r_2)$ has 2 parts, we will designate R_1 and R_2, of sizes r_1 and r_2 respectively. These have at least one

vertex in either part. Unless the graph $H_2(r_1, r_2)$ is a complete multipartite graph then $G_4(g)$ must be locatable since the remaining two vertices can be put into a locating row with x and y. If $H_2(r_1, r_2)$ is complete then there must be at least $|r_1 r_2|$ edges, this is minimal when both parts are equal. However, since the maximum degree of any vertex is $\frac{g-1}{2}$ and the smallest size of any part $r_i \leq \frac{g+1}{2}$ then the size of our subgraph $|E(H_2(r))| \leq \left(\frac{g+1}{2}\right)\left(\frac{g-1}{2}\right)$. Which is not sufficient to create a complete multipartite graph on $H_2(r_1, r_2)$. Thus it is possible to put the vertices x and y into an independent 4-tuple, therefore $G_4(g)$ is locatable. □

We can again create a similar algorithm to create a locating row on a given $G_4(g)$.

Algorithm: 2.5.3 a locating Row Algorithm on a graph $G_4(g)$

Begin Algorithm

Function $G(g_1, g_2, g_3, g_4)$ **Locater** (G, x, y)

\\Inputs a multipartite graph G with 4 parts and two vertices x and y

\\in different part of G

$S = \{x, y\}$

$G = G \setminus M(x)$

$G = G \setminus M(y)$

For $(i = 1$ **to** $2)$

Sort(R_1, R_2) \\Here we sort parts from smallest to largest

If $|R_1| = 0$ \\If the smallest part is 0 the algorithm has failed

Return *Fail*

$a =$ **Random Select**(R_1) \\Selects a random vertex from R_1

$S = S \cup \{a\}$

$G = G \setminus M(a)$

Next i

Return S

End Algorithm

Chapter 3

General Degree Bounds for Locatability

3.1 Introduction

As previously mentioned, although it is possible to derive tight degree constraints for some special graphs, we would like to take a more general approach and explore degree bounds for an arbitrary testing problem TP. We look at one way to determine locatability on a graph with arbitrary factors. In addition, a greedy algorithm can be used to create locating rows and we show this algorithm is guaranteed not to fail given certain degree constraints. We also discuss a bound derived from an application of a modified version of

Turán's theorem. It should be noted that while the construction of locating rows is in general an NP-Complete problem, as shown in Theorem 1.4.4. However, given certain degree constraints it is possible to construct locating rows in polynomial time.

3.2 A Turan-type proof

We normally consider failing interactions to be edges of the error graph but in a complementary problem we may consider the non failing interactions to be the edges in our graph and attempt to build a k-clique on those edges. We can consider vertex degrees in the error graph to be non-edges in a Turán graph. We previously discussed Turán results in Section 1.2.4 and we will need some of the tools developed in that section here. The problem then becomes how to put 2 vertices into a clique with one vertex from each part of our graph. In this formulation, upon deletion, we will allow each vertex to delete Δ edges from our graph. We then ask how many edges can each vertex delete while still guaranteeing a clique can be formed. Cliques have the useful property that, on a multipartite graph $G(g_1, \ldots, g_k)$, any clique of size k must contain one vertex from each part.

Fortunately, we can use some pre-existing work to help us generate a bound on the number of allowed non edges in a graph $G_k(g)$ while still forcing a clique spanning all parts to exist between any two vertices. Nagy

[14] has shown the following result.

Theorem 3.2.1 (Turan Based Bound [14])**.** *Let G be a graph G of order k and size M. If G is not contained in its blow up graph $G_k(g)$ then $G_k(g)$ has at most $(M-1)n^2$ edges.*

We now have a bound on the most edges a blow up graph of G can have without it containing a copy of G. We can take this bound and use it to examine some limitations on the general error graph, $G_k(g)$.

Theorem 3.2.2 (Bound on edge deletions in a multipartite graph)**.** *If a multipartite graph $G_k(g)$, with $k > 4$, has degree $\Delta \leq \lfloor \frac{2g}{k} \rfloor$ it must contain an independent set between any two vertices containing one vertex from each part.*

Proof. Given two vertices x and y we can analyze the graph after we have removed $M(x)$ and $M(y)$. Since we only need a clique on the remainder $R_k(g) = G_k(g) \setminus (M(x) \cup M(y))$ we then analyze $R_k^*(g)$.

Using Theorem 3.2.1 with our blow up graph $H = K_{k-2}$, if we have $\binom{k-2}{2}g^2$ edges in our blow up graph then we are guaranteed to have a clique of order $k-2$ somewhere in $G_k(g)$. When taken with x and y this clique would form a locating row on the error graph. Thus if $\left(\binom{k-2}{2} - 1\right) g^2 \geq |E(R_k^*(g))|$ then $\exists\ K_{k-2} \subseteq R_k^*(g)$.

Since, we have inverted our usual error graph we will be deleting edges

from a complete multipartite graph. Therefore, the total number of edges is

$$|E(R_k^*(g))| = \binom{k-2}{2}g^2 - \frac{gk\Delta}{2}.$$

Noting that, this problem reduces to

$$\binom{k-2}{2}g^2 - \frac{gk\Delta}{2} \geq \left(\binom{k-2}{2} - 1\right)g^2.$$

When $k > 4$ this becomes

$$\Delta \leq \frac{2g}{k}.$$

Since Δ is an integer, we may write the bound as $\Delta \leq \left\lfloor \frac{2g}{k} \right\rfloor$. □

Of course this theorem would predict that for $k > 2g$ that $\Delta = 0$. However, we know by Theorem 2.4.1 that given a graph $G(g_1, \ldots, g_k)$ where every $g_i \geq 3$ and $\Delta = 1$ it will be locatable for any k, so this bound cannot be tight.

3.3 An Algorithm for General k

We start with a k-partite graph $G(g_1, \ldots, g_k)$, we are attempting to put any 2 vertices x and y from different parts into an independent set with one vertex from each part. We can remove $M(x)$ and $M(y)$ from our graph to get an new graph $G(r_1, \ldots, r_{k-2})$. We are trying to create a locating row

S containing x and y, so we add x and y to S. We now iterativly pick the smallest part, choose a vertex from that part to add to our set and delete the Meighbourhood of that vertex. If at any step the smallest part is empty the algorithm fails. The proof of Theorem 3.4.1 relies on showing that if a certain degree bound is met, this algorithm never fails.

Alogrithm 3.3 Input a multipartite graph G with k parts and pick 2 vertices x and y from each part.

Begin Algorithm

Function **Greedy Locate** (G, x, y)

$S = \{x, y\}$

$G = G \setminus M(x)$

$G = G \setminus M(y)$

While $(|S| <= k)$ \\Once we have a set of k vertices our algorithm \\exits successfully

Sort$(R_1, \ldots, R_i)$ \\We sort our remaining graph by parts from \\smallest to largest

If $|R_1| = 0$

Return *Fail* \\If at any point our algorithm encounters \\an empty part

it fails

$a =$ **Random Select**(R_1) \\Selects a random vertex from R_1

$S = S \cup a$

$$G = G \setminus M(a)$$

End while

Return S

End Algorithm

If this algorithm loops for $k-2$ times then we must have our independent set since S will now contain k vertices.

3.4 Establishing Bounds for a general graph $G_k(g)$

While the Turán bound examined in Section 3.2 provides a useful estimate, a different approach arises when we examine a graph $G_k(g)$ and we carefully choose vertices in parts when creating a locating row. We do this by selecting the smallest part after each progressive deletion and then choosing a vertex at random in that part, v_i and deleting the meighbourhood set of that vertex $M(v_i)$. If this can continue this process for $k-2$ vertices then we can be sure that the set of all vertices we selected originally is independent.

This raises an immediate question about how to assign the edges of each selected vertex. We want to consider the "worst case" configuration for any graph. The "worst case" in our construction is the case where after i itera-

tions a graph $G_k(g)$ has the fewest number of vertices remaining. However if any part becomes smaller than all the others, our algorithm will remove that part first. So when we calculate the bound of our algorithm we presume that at each step the remaining parts are having vertices deleted evenly. This is not always possible since we will not necessarily have a multiple of Δ parts at each step. However it serves as a useful lower bound to work from.

Theorem 3.4.1 (Degree bound for a locatable graph)**.** *Given a multipartite graph $G_k(g)$, with $k > 4$ and every vertex having maximum degree*

$$\Delta \leq \left\lfloor \frac{g-1}{\frac{2}{k-2} + \left\lfloor \sum_{j=1}^{k-3} \frac{1}{k-2-j} \right\rfloor} \right\rfloor .$$

For any two vertices x and y there is a row that locates x and y. That is this graph is in AVOID(2).

Proof. By Theorem 2.1.1 we need only consider the regular degree graph. We will put x and y into a locating row, that is we will find a set of independent vertices one from each of the remaining parts. First we delete the meighbourhood sets of x and y, $M(x)$ and $M(y)$. We have deleted at most

$$\frac{2(g-1)}{k+1} + \frac{2(g-1)}{k+1} = \frac{4(g-1)}{k+1}$$

vertices from all parts.

We now iterativly choose a "live" vertex in the smallest remaining part,

add it to our set and delete it's meighbourhood. At the i^{th} step let us define the set of the remaining vertices in each part as $R_i, \ldots, R_{k-2-i}$. Without loss of generality we take $|R_1| \leq |R_2| \ldots \leq |R_{k-2-i}|$. We note that at the first step $i = 0$, $|R_1|$ cannot be 0 since $\Delta < g/2$. In general, $|R_1|$ cannot be greater than the average number of deletions over all remaining parts. Since initially there are $k-2$ parts remaining,

$$|R_1| \leq \frac{g(k-2) - 2\Delta}{k-2}.$$

We now pick some $a_1 \in R_1$, we add this to our set S and delete $M(a_1)$. After every iteration i we lose at most Δ vertices from our remaining graph with parts $\mathcal{R} = \{R_1, R_2, \ldots, R_{k-2-i}\}$.

Since on the first step $|R_1| \geq 1$ we can't eliminate that part on the first round and if all edges are allocated to R_1 then all those edges will be removed at the next step. However, if we spread the edges evenly across 2 parts we will only lose $\frac{\Delta}{2}$ edges in the next iteration. Recursively, the strategy which leads to the *fewest* edges being removed at each iteration is to spread the edges over Δ parts if $n > \Delta$, where n is the number of remaining parts and $\frac{\Delta}{n}$ otherwise i.e. Thus we want to minimize the number of vertices remaining, so we delete the average number of vertices on each part at each step. Thus the smallest part remaining, R_1, would have received the floor of the average number of deletions. After i steps the average number of deletions on each

part is

$$\left\lfloor \sum_{j=1}^{i} \frac{\Delta}{k-2-j} \right\rfloor .$$

As long as our smallest part $|R_1| \geq 1$ there is some vertex that is independent of all previously chosen vertices $\{x, v_1, \ldots, v_i, y\}$. After $k-3$ steps we have one remaining part so long as that part $|R_1| \geq 1$ a locating row can be constructed. This can be represented by the inequality

$$\frac{g(k-2)-2\Delta}{k-2} - \Delta \left\lfloor \sum_{j=1}^{k-3} \frac{1}{k-2-j} \right\rfloor \geq 1.$$

Where the left hand side represents the size of the last remaining part. This can be expressed in terms of Δ as

$$\Delta \leq \left\lfloor \frac{g-1}{\frac{2}{k-2} + \left\lfloor \sum_{j=1}^{k-3} \frac{1}{k-2-j} \right\rfloor} \right\rfloor .$$

□

Although this bound is interesting it is by no means the lowest possible bound. We might want to use a different more simpler bound instead of the complex one we currently have for this we can create a corollary which is simpler.

Corollary 3.4.2. *If a given graph $G_k(g)$ has degree $\Delta \leq \left\lfloor \frac{g-1}{\left\lfloor \frac{k+1}{2} \right\rfloor} \right\rfloor$, it is locatable.*

Proof. If $k = 3$ the bound is established by Theorem 2.5.1 and if $k = 4$ the bound is established by Theorem 2.5.2. Therefore we now presume $k \geq 5$ and use Theorem 3.4.1 by showing that

$$\left\lfloor \frac{g-1}{\frac{2}{k-2} + \sum_{j=1}^{k-3} \frac{1}{k-2-j}} \right\rfloor > \left\lfloor \frac{g-1}{\lfloor \frac{k+1}{2} \rfloor} \right\rfloor.$$

Since we can use the integral approximation of our sum to derive

$$\sum_{j=1}^{k-3} \frac{1}{k-2-j} = 1 + \sum_{j=1}^{k-4} \frac{1}{k-2-j} \leq 1 + \int_1^{k-3} \frac{dx}{k-2-x} = 1 + \ln(k-3).$$

Now consider the inequality

$$\frac{2}{k-2} + 1 + \ln(k-3) = \frac{k}{k-2} + \ln(k-3) \leq \left\lfloor \frac{k+1}{2} \right\rfloor.$$

This inequality holds for $k = 5$ and the right hand side grows linearly, while the left hand side grows logarithmically, therefore it is easy to see that this inequality holds for $k > 5$.

Since we have shown

$$\left\lfloor \frac{g-1}{\frac{2}{k-2} + \left\lfloor \sum_{j=1}^{k-3} \frac{1}{k-2-j} \right\rfloor} \right\rfloor \leq \left\lfloor \frac{g-1}{\frac{2}{k-2} + ln(k-3)} \right\rfloor \leq \frac{g-1}{\lfloor \frac{k+1}{2} \rfloor}.$$

Therefore the result follows.

□

This approach is obviously not exact, the approach is entirely greedy in the sense that all vertices chosen are chosen at random. It is possible that we could allow for a higher bound if we used a method for constructing locating rows for all pairs of vertices that do not rely on random selections. Indeed, as k becomes large these bounds become 0. However, Theorem 2.4.1, will guarantee the construction of a locating row for $\Delta = 1$. Still, for $g >> k$, this approach guarantees the construction of some locating row on a graph with degree $\Delta > 1$.

3.5 Creating an Error Locating Array

Previously we have only discussed algorithms which create individual locating rows given any two test vertices. While in general this is sufficient for showing that a given graph is locatable, ideally we would like construct an algorithm where we build an array which covers every pairwise interaction in a given graph G. We can create an algorithm which calls one of our other algorithms 2.4.1, 2.5.2, 2.5.3, or 3.3. When this is done we produce a Error Locating Array.

We will be adding locating rows to an array L while minimizing the number of rows in the array in total. We keep track of the coverage between

pairs x and y throughout, with covered(x,y) which is true if a pair xy has been covered and false otherwise. This is so we don't have any extra rows. We call the function **Row** which will be defined by the type of graph we start with initially, in practice one of the algorithms described above. The function **addrow** is used to add the Row S to the array L. So long as every pair x and y can be put into a locating row, this algorithm will not fail.

Algorithm: 3.5 ELA construction algorithm for a given graph G

Begin Algorithm

Function ELA Creator (G, k, x, y, Row)

for all pairs $x, y \in G$ set covered(x, y) = *False*

$L = \emptyset$

While (there is an uncovered pair, x, y)

S = Row(G, k ,x, y)

if S = *Fail*

Return *Fail*

for (every pair, $s_1, s_2 \in S$)

covered(s_1, s_2) = *True*

L = **addrow**(L,S)

End While

Chapter 4

Conclusion

4.1 Results

Locatability was the primary motivation of this work. Viewing the general testing problem as a graph theoretic one was the lens through which locatability was considered. Traditionally the testing problem focuses on the total number of failing interactions in a given TP. However, by viewing those errors as an edge in a graph one begins to consider the degree, a basic property of a graph.

Degree Bounds

The first aspect of this graph theoretic approach to be realized was made in Theorem 2.1.2, that if any partite subgraph is non locatable the whole graph is non-loctable. This lead to the conclusion that the degree bound $\Delta = \left\lfloor \frac{g-1}{2} \right\rfloor$ is sharp, even one more failing interaction could create a non-locatable 3-partite subgraph and thus no graph $G_k(g)$ can have a higher degree and guarentee locatability. In addition, the same bound held for a 4-partite graph $G_4(g)$. Interestingly, no non-locatable graph with degree bound $\Delta \leq \left\lfloor \frac{g-1}{2} \right\rfloor$ was ever characterized. There is however a lower bound for a graph with any number of parts k. If any graph $G(g_1 \dots g_k)$ with all $g_i \geq 3$ has a degree $\Delta = 1$ then that graph is certainly locatable, and locating rows can be constructed in linear time. A general result was obtained using a variant of Turáns theorem, described by Nagy [14]. This result establishes a lower degree bound of $\Delta = \left\lfloor \frac{2g}{k} \right\rfloor$ and although this is not a sharp lower bound it does give a maximum degree for some graph $G_k(g)$.

A known bound for any graph with arbitrarly many parts k,

$$\Delta \leq \left\lfloor \frac{g-1}{\frac{2}{k-2} + \left\lfloor \sum_{j=1}^{k-3} \frac{1}{k-2-j} \right\rfloor} \right\rfloor,$$

is derived Theorem 3.4.1. In Corollary 3.4.2 we derive the bound $\Delta = \left\lfloor \frac{g-1}{\left\lfloor \frac{k+1}{2} \right\rfloor} \right\rfloor$, for this bound an algorithm is described which can construct locating rows in linear time. In fact in all cases discussed an algorithm was

described which will return a locating row in linear time if given the appropriate bounds.

We have outlined some special cases where it is possible that all graphs created will be locatable. However, all of these cases are rather specific and all the algorithms require foreknowledge of the structure of the error graph. That is, they require that the neighborhoods of all vertices be known in advance before any actual locating rows can be constructed. However, in a practical case this might not be as limiting as one thinks. If enough locating rows can be constructed initially, that information can be used to map out many of the interactions which are not failing interactions. When this is done, those interactions become effectively the non-edges of our error graph. With this information a more complete picture of the error graph can be created, which might allow for the construction of further locating rows more directly.

While we may be able to find specific classes of graphs which are locatable or non-locatable. Locatability is in general an NP-Complete problem. Therefore it is unlikely that we will be able to find a general algorithm constructing locating rows for an arbitrary graph $G_k(g)$.

4.2 Conjectures and Possible Future work

Conjectures

In this section we will briefly discuss some conjectures that arose during the creation of this thesis.

Conjecture 4.2.1. There is some fixed value l, for which non-locatable $H_l(g)$ (or a set of non-locatable $H_l(g)$) exists such that for any non-locatable graph $G_k(g)$, with $\Delta = \frac{g-1}{2}$ and $k > l$, the remaining induced subgraph $G'_j(g) = G_k \setminus H_l(g)$ is locatable.

The bound $\Delta \leq \frac{g-1}{2}$ is a sharp upper bound, even one more edge can form a non-locatable graph on three parts. It remains open however whether or not given this restriction non-locatblility is truly global or whether there is some number of parts beyond which any non-loctable interaction must be on a partition subgraph.

Conjecture 4.2.2. A graph $G_k(g)$ is locatable if it has maximum degree $\Delta \leq \frac{g-1}{2}$.

This arises from the fact the we were unable to construct a non-locatable graph $G_k(g)$ for *any* k, with $\Delta \leq \frac{g-1}{2}$ and although we could not conclusively prove that this bound holds in general, we could not come up with a counter example. It should be noted that this bound is k independent. Even if con-

jecture is false there may be a lower bound which is k independent. Indeed, the bound $\Delta = 1$ found in Theorem 2.4.1 is such a bound.

Conjecture 4.2.3. There are no P-time algorithms for constructing locating rows for a general graph $G_k(g)$ with a degree $\Delta = \frac{g-1}{2}$.

In general,, constructing a locating row is NP-Complete. Although we characterized some algorithms which can construct locating rows in P-time, they did not use the bound $\Delta \leq \frac{g-1}{2}$ except on $G_3(g)$ and $G_4(g)$. In fact using $\Delta \leq \frac{g-1}{2}$ as a bound could cause the general Algorithm 3.3 to fail on a $G_5(g)$ graph even though this graph may be locatable.

Further Work

We now discuss some future work that might be done in this field. The study of testing problems as a mathematical object is relatively new, and thus has not been well explored. Particularly, graph theoretic approaches still have many unexplored facets which could yield interesting new approaches to creating ELA's.

Although work exsists on the relationship between the size of Detecting Arrays and the number of errors in a given TP [5], we did not investigate the impact that degree constraints would have on the minimum size of a detecting array. We dismissed "randomly" constructing an array which correctly locates errors Section 1.1.2 as unreliable. However, it remains open whether

or not some "probablistic" method exists which can create ELA's which are not guaranteed to locate all errors but are likely to. Of particular interest would be the possibility of creating smaller ELA's with this method.

Although all the algorithms we described searched for pairwise interactions. It should be possible to construct locating rows for any strength t interactions. A good focus for future work might also be the construction of mixed strength algorithms which can create rows which locate up to t strength interactions.

As an applied topic, a survey of where failing interactions lie in real world testing problems would be interesting. Although we can set bounds where non locatable graphs can be constructed. Very particular edge configurations are required to actually create non-locatable interactions. It would be interesting to learn how probable these configurations actually are.

Bibliography

[1] S. Arora, B. Barak *Computational Complexity: A Modern Approach*, Cambridge University Press, 2009

[2] R. Bryce, C. Colbourn, *A density-based greedy algorithm for higher strength covering arrays*, Software Testing, Verification and Reliability, 2009, **19**: 37–53

[3] C.J. Colbourn, D. McClary *Locating and detecting arrays for interaction faults* Journal of Combinatorial Optimization (2008), **15**: 17–48

[4] C.J. Colbourn, G. Keri, P.P. Rivas Soriano, J.-C Schlage-Puchta, *Covering and radius-covering arrays: Constructions and classification* Discrete Applied Mathematics **158**, 2010,

[5] C.J. Colbourn, Covering Array Tables. *http://www.public.asu.edu/ ccolbou/src/tabby/catable.html*

[6] P. Danziger, E. Mendelsohn, L. Moura, B. Stevens, *Covering Arrays Avoiding Forbidden Edges*, Theoretical Computer Science **410**: 5403-5414, 2009

[7] C.Colbourn, J. Dinitz *Handbook of Combinatorial Designs (2nd ed.)* CRC Press, 2007

[8] L. Gargano, J. Korner, U. Vaccaro *Sperner Capacities* Graphs and Combinations, **9(1)**: 31-46, 1993

[9] A. Hartman, L. Raskin, *Problems and algorithms for covering arrays*, Discrete Mathematics **284(1–3)**: 149–156, 2004

[10] R. Kuhn, M. Reilly, *An Investigation of the Applicability of Design of Experiments to Software Testing* Proceedings of the 27th Nasa Software Engineering Workshop, 2002

[11] E. Maltais, *Covering Arrays Avoiding Forbidden Edges and Edge Clique Covers* Master's Thesis, Master of Science in Mathematics, University of Ottawa

[12] K. Meagher, B. Stevens, *Covering arrays on graphs* Journal of Combinatorial Theory Series B, **95(1)**: 134-151 2005

[13] E. Maltais, L. Moura *Hardness results for covering arrays avoiding forbidden edges and error-locating arrays* Theoretical Computer Science, **412(46)**: 6517-6530, 2011

[14] Z. Nagy, *A multipartite version of the Turan problem-density conditions and eigenvalues* The Electronic Journal of Combinatorics **18(1)**, 2011

[15] C. Martínez, L. Moura, D. Panario, B. Stevens, *Locating Errors Using ELAs Covering Arrays, and Adaptive Testing Algorithms* Society for Industrial and Applied Mathematics, **23(4)**: 1776–1799, 2009

[16] H. Moskowitz, S. Porretta, *Concept Research in Food Product Design and Development* Wiley, 2005

[17] Y. Tang, C. Colbourn, J. Yin *Optimality and Constructions of Locating Arrays* Journal of Statistical Theory and Practice, **6**: 20-29 2012

[18] S. Martirosyan, T. Van Trung, *On t-Covering Arrays*, Designs, Codes and Cryptography, **32**: 323 2004

[19] D. West, *Introduction to Graph Theory 2nd ed.*, Pearson, 2001